A Distributed Implementation of Continuous-MapReduce Stream Processing Framework

Zikai Wang
Department of Computer Science, Brown University
zikaiwang@cs.brown.edu

May, 2011

1 Introduction

Stream processing frameworks exploit data, task and pipelined parallelism to minimize end-to-end latency which is crucial for their application. To achieve this goal and overcome difficulties that lie naturally in streaming workloads: varying data complexity, data skew, volatile stream rates and bursty streams, they have to efficiently utilize parallel computing resources. Traditionally, stream processing engines assign operators to computing nodes statically and such assignment is resilient to volatility. As a result, load-balancing choices are seriously limited which causes bad node utilization, workflow-wide load imbalances and eventually high end-to-end latency.

Continuous MapReduce(C-MR) [1,2] is a new stream processing framework that enables continuous execution of MapReduce jobs on data streams. C-MR introduces both ideas of windowing constructs and generic computing nodes to allow for pipelined data consumption, incremental processing and reduction in redundant computations for shared input streams. Meanwhile, generic computing nodes allows performing fine-grained load balancing for heterogeneous architectures and aggressive scheduling policies which are capable of using the collective computing power of all available nodes to target specific problem areas.

This report introduces the work of extending C-MR from a single-host system (denoted as C-MR-S) to a fully distributed system (denoted as C-MR-D) that utilizes computing capacity of clusters. Section 2 provides an introduction on backgrounds of C-MR processing model, C-MR-S and design concerns of C-MR-D. Section 3 describes the high-level architecture of C-MR-D and its distributed system mechanisms as well as low-level architecture of each instance. Section 4 describes implementation details of instance modules that are related to the distributed system mechanisms.

To clarify the terminology in this report, a 'node' always refers to a 'generic computing node' which is the basic unit of computing resources and usually a CPU core or a GPU on a physical host. A 'host' refers to a physical machine that runs a C-MR-D instance. An 'instance' is an invocation of C-MR-D binary on a host. An instance may have multiple nodes depending on number of processors/cores/GPUs available on the invoking host. Instances communicate with each other and work together as a C-MR-D cluster.

2 Background

2.1 C-MR Processing Model[1]

C-MR processing model is an extension to the traditional MapReduce processing model. In order to execute over unbounded streams of data continuously and efficiently, C-MR processing model deviates from the traditional MapReduce processing model by introducing concepts of windowing and generic computing nodes, eliminating disk buffers, enabling pipelined processing and incremental processing.

C-MR processing model introduces the concept of windowing to support aggregation on unbounded streams. By dividing a stream into sets of temporarily contiguous data items called windows, operations are be evaluated over them and the results are propagated along the workflow. Windows are defined by a size and a slide. The size defines boundaries of the temporally contiguous set included in the window and the slide determines the interval at which windows are defined and evaluated. C-MR processing model supports creation of windows based on system timestamp as well as application timestamp.

Generic computing nodes are computing resource units that are capable of working on any map/reduce task in the workflow at any time. They could be physical cores of CPUs as well as GPUs. They typically asynchronously extract data from intermediate buffer, execute the corresponding task and place the results in a downstream buffer. By requiring fully materializing windows before nodes start processing reduce tasks, computing nodes never have to wait for stragglers to finish a corresponding set of tasks when there is data else-where waiting to be processed. Introduction of generic computing nodes enables C-MR processing model to perform pipelined processing in which a window for any key can be processed at any downstream computing node as soon as it is materialized regardless of whether other windows have been materialized. This is a sharp contrast to traditional MapReduce which batches data between map/reduce phases and requires strict assignments of computing nodes to tasks.

Traditional MapReduce relies on disk to buffer intermediate results and introduces high latency overhead which is unacceptable for stream processing. Therefore, C-MR processing model instead relies on shared-memory buffers accessible by all computing nodes on the same host to store intermediate results. As data migrates through the logical workflow between computing nodes, only control of references is passed.

C-MR processing model also adopts incremental processing of each reduce window to avoid high latency of batching it entirely before it could be processed. This done by decomposing each reduce window into sub-windows and perform a Combine operation on them.

2.2 Single-Host C-MR (C-MR-S)[1]

C-MR-S is a single-host implementation of the C-MR processing model which has all features described in Section 2.1. C-MR-S uses physical cores and GPUs of the host as computing nodes. It adopts an asynchronous, push-based processing model which allows computing nodes to execute any map/reduce tasks as they are received. Computing nodes communicate with each other via shared memory.

2.2 Distributed C-MR (C-MR-D)

C-MR-S is limited by capacity of a single host. To fully utilize power of C-MR processing model, it is natural to extend C-MR-S to a distributed system that leverages a cluster of multi-core and multi-GPU hosts. Therefore, we design and implement C-MR-D, a distributed implementation of C-MR processing model on top of C-MR-S. C-MR-D uses C-MR-S as building blocks that run on individual host and handle computation on a single partition of data. On top of C-MR-S, we build an additional layer to handle distributed system mechanisms like data flow, control flow, data partitioning, load balancing and membership protocol. C-MR-S is modified properly to work seamlessly and efficiently with that layer.

C-MR-D is targeted at deployment in an environment with shared-nothing, heterogeneous infrastructure. In a shared-nothing infrastructure, each host has its own independent CPU(typically multi-core), memory, disk and GPU. In a heterogeneous infrastructure, different hosts may have distinct computing resources (CPU and GPU) and storage resources (memory and disk). Also, there are a number of generally desired properties for a distributed computing system including scalability, availability and fault-tolerance. Furthermore, our specific application area of stream processing requires minimizing end-to-end latency. The target deployment environment and various desired properties bring us a series of design concerns.

As an example, to achieve latency efficiency in a heterogeneous environment, workload partitioning strategy has to be fine-grained and adaptive to volatile workload, host capacity and changing host load. Another example is that a non-traditional K-safe variant of upstream backup is needed to provide both fault-tolerance and low latency

1 Introduction on C-MR processing model and C-MR-S in Section 2.1 and 2.2 are based on [1, 2]

overhead.

We try to take these design concerns into account when laying out architecture of the system and pursue both general and application area specific desired properties. At current preliminary version of C-MR-D, we have a fully functional distributed system and we have good scalability and performance. We are currently exploring strategies of workload partitioning and load balancing and leave some features like fault-tolerance to be implemented in future work.

3 Architecture

In general, C-MR-D embraces a decentralized, symmetric architecture optimized for heterogeneity in the infrastructure. Data is horizontally partitioned over all instances. Each instance has the same responsibility in terms of processing the work flow as its peers. However, the amount of data processed at a specific instance depends on its capacity and current load. Load balancing decisions are made at operator level dynamically. A special instance called the Master is responsible for making such decisions based on latest load statistics collected from all instances.

We expect the Master will not become a performance bottleneck because we separate data flow and control flow and minimize its load. As a data processing node as any other instance, the Master will always handle a proper amount of data proportional to its capacity and load. This is guaranteed by the load balancing mechanism. Meanwhile, though it also plays essential roles in control flow like collecting load statistics from other instances, making load balancing decisions and issuing new routing tables, the amount of data in the control flow is several orders of magnitude less than that in the data flow.

In the following sub-sections, we will introduce core distributed system mechanisms in detail including data flow, control flow, load balancing and membership protocol.

3.1 Data flow and control flow

In C-MR-D, data flow is determined by routing tables at operator level. A routing table horizontally partitions input data for an operator over all instances and reflects the latest load balancing decisions by the Master. Each operator has its own routing table independently created and updated. Initial input data for the first operator(s) in the workflow comes from an external source while for other operators input data comes from running C-MR-D instances. Initial input data is sent to the Master, who routes it to all instances based on the initial uniformly partitioned routing table. When an instance receives data for an operator, the data will be processed as in C-MR-S. When the instance finishes processing that data, it will look up in the latest routing table for the next operator in the workflow and route the result data to a proper instance. That result data is used as input data for the next operator in the workflow. When data for an operator at the end of the workflow is processed and a final result is generated, it is routed to the Master which collects and reports final results of the stream processing task.

Currently, control flow is simple. There are only three types of control messages: load statistics, new routing tables and punctuations. Part of the reason is that we are using a static membership protocol now. We expect to have more types as we extend the membership mechanism to allow dynamic instance additions and removals.

3.2 Statistics Collection and Load Balancing

To support load balancing decision making, each C-MR-D instance is actively collecting load statistics. Currently, we use latency as main criteria to measure load of an instance at operator level. At a certain instance, for each operator and each data item processed by that specific operator, we measure the sum of queuing time and processing time as latency. We maintain average latency for each operator at each instance. C-MR-D instances periodically report such statistics to the Master.

The Master will periodically make load balancing decisions based on available load statistics. Ideally, these decisions should take loads and capacities of instances as well as locality into account. Currently, we are still exploring tradeoffs between different strategies. The Master's load balancing decisions are expressed as new routing tables, which determine data partitioning for each operator. New routing tables are broadcasted to all C-MR-D instances. They will apply the new tables later at a proper time. Note that it is possible that new routing tables for different operators are generated at different time.

3.3 Membership Protocol

Currently, C-MR-D uses a static membership protocol and instances initialize membership information from a static instances list. The list specifies host and port of each instance. When a C-MR-D cluster is launched, an instance is started at each host. Then in a period called connection establishment, C-MR-D instances try to establish pair-wise TCP connections. Once all connections are built successfully, the cluster is ready to process incoming data stream.

Incremental scalability and elasticity are important for a distributed computing framework, we plan to extend C-MR-D membership protocol for dynamic instance additions and removals in the future. Candidates for such protocol include Chord[3] and Gossip-based protocol in Dynamo[4].

4 Implementation Details

C-MR-D is implemented in C++ and compiled with GCC. A C-MR-D instance has four main modules that handle distributed system mechanisms: Input Manager,
Output Manager, Statistics Manager and Network Manager. Each of these modules is encapsulated in a C++ class. We are going to describe their functionalities, essential data structures and major public methods in the following sub-sections.

4.1 Input Manager

Input Manager is the module responsible for maintaining local C-MR-D instance's unique input queue. The input queue is coupled with two threads, the TCP listener thread and the input thread.

The TCP listener thread handles incoming network traffic in a synchronous I/O multiplexing mode provided by select(). It monitors TCP connections to other C-MR-D instances. When there is an incoming message at one of the TCP connections, it will determine type of the message, deserialize it, then take proper action. For example, a data item will be inserted into the input queue waiting to be processed by a generic computing node while a new routing table created and sent by the Master will be applied later at a proper time. The TCP listener thread also passively listens to a service specific port and accepts TCP connections from other C-MR-D instances at connection establishment period when all instances are trying to connect to each other before they could process any data.

The input thread works closely with the input queue. Generally, when the TCP listener thread receives and deserializes a data item, it is inserted into the input queue. The input queue is a blocking queue. When it is empty, the input thread is blocked. Otherwise, the input thread will dequeue a data item and hand it to the Workflow Manager, which schedules execution of the workflow. Eventually, the data item will be processed at a generic computing node.

4.1.1 Data Structure

• bqueue_t inputQueue
The input queue 'inputQueue' is a blocking queue. Any type of item could be inserted as long as pointer to it is provided

• pthread_t nodeThread
'nodeThread' stores thread ID of the input thread

• pthread_t listenThread
'listenThread' stores thread ID of the TCP listener thread

4.1.2 Public Methods

• void enqueue(DataIterator* di)
Insert a data item or a punctuation into the input queue. Note that the timer of the data item for measuring latency is started here

4.2 Output Manager

Output Manager is the module responsible for maintaining local C-MR-D instance's unique output queue. Generally, when a generic computing node finishes processing some data, the results are inserted into the output queue. The output queue is a blocking queue and coupled with an output thread. When it is empty, the output thread is blocked. Otherwise, the output thread will dequeue a data item. The data item is routed to a proper C-MR-D instance according to corresponding operator's current routing table. Therefore, output queue (and output thread) effectively connects data flow between dependant operators on the same instance or different instances.

There are two other types of items that could be inserted into the output queue: puncutations and statistics. We make them handled by the output queue (and output thread) rather than creating individual functions to send them out to make the design more elegant. Normally, punctuations are broadcasted and statistics are sent to the Master.

4.2.1 Data Strctures

• bqueue_t outputQueue
The output queue 'outputQueue' is a blocking queue. Any type of item could be inserted as long as pointer to it is provided

• pthread_t nodeThread
'nodeThread' stores thread ID of the output thread

4.2.2 Public Methods

• void enqueue(DataIterator* di)
Insert a data item or a punctuation into the output queue. Note that as described in Section 4.3, the timer of a result data item is stopped before it is insrted into the queue, latency is then extracted and average latency in Statistics Manager is updated

4.3 Statistics Manager

Statistics Manager is the module responsible for maintaining load statistics on local C-MR-D instance. Currently, we use latency as main criteria to measure load of an instance at operator level. To be more specific, for each operator and each data item processed by that specific operator, we measure the elapsed time or latency from that data item enters input queue to it gets processed by a generic computing node and the result data item is sent to output queue. At an implementation level, latency is measured by attaching a timer to each data item. For a map operator, the timer is started when the input data item enters input queue of Input Manager. It is propagated properly all the way to the result data item generated by map function at a generic computing node. It is stopped when the result data item enters output queue of Output Manager. Latency is then extracted and the statistics are updated. For a reduce operator, way of measurement is similar except that at a reduce operator, a window of data items are batched and processed together and one data item is generated as result. In this circumstance, timer of the punctuation that materializes the window is propagated to the result data item. For both cases, the measured latency captures both queuing time and processing time.

For each operator, we maintain the moving average of latencies on all data items. C-MR-D instances will periodically report load statistics to the Master who makes load balancing decisions based on statistics collected from all nodes and issues new routing tables.

4.3.1 Data Structures

- vector<Average> avgProcessingTime

Average latency table 'avgProcessingTime' stores average of latencies on data items processed by each operator. IDs of operators are used as indexes for looking up in the table. Average is calculated as moving average, which is encapsulated in class Average

4.3.2 Public Methods

- double updateAverage(DataIterator* di)

This method is called when a result data item is about to enter the output queue. It will stop the timer attached to the data item, extract the latency and update statistics. Currently, it will also send latest statistics of local C-MR-D instance to the Master

- double getAvgProcessingTime(uint16_t opID)

Get average latency of an operator with a specific ID

4.4 Network Manager

Network Manager is the module responsible for keeping membership information including addresses of running C-MR-D instances and TCP connections to them. It also maintains routing tables for operators that are essential to data partition and load balancing. More specifically, the routing table for a certain operator records current partition of key hash space to continuous sub-regions and assignment of these sub-regions to instances. Data is partitioned, routed to proper instances then processed there according to routing table of the corresponding operator. The Master is responsible for making load balancing decisions based on load statistics collected from all nodes and issuing new routing tables.

Network Manager also encapsulates a series of important methods for data flow and control flow including sending latest routing table to all C-MR-D instances, routing data to proper C-MR-D instance, sending load statistics to the Master, broadcasting a punctuation and so on. These methods are used by other modules like Output Manager and Statistics Manager.

4.4.1 Data Structures

• map<uint32_t, CmrNodeAddr*> nodeAddrTable
The node address table 'nodeAddrTable' stores the mapping from ID of C-MR-D instances to their addresses. 'CmrNodeAddr' is a struct that contains hostName and port of an instance

• map<uint32_t, TCPConnection*> tcpConnection
The TCP connection table 'tcpConnection' stores the mapping from ID of C-MR-D instances to TCP connections to them. 'TCPConnection' is a class that encapsulates TCP connection and communication

• map<uint16_t, RoutingTable*> routingTable
The routing table 'routingTable' stores the mapping from ID of operators to their individual routing tables. Each operator has its own routing table because we would like to perform a fine-grained load balancing at operator level. Type 'RoutingTable' is map<uint32_t, uint32_t> in which keys are hashes of data items and values are IDs of C-MR-D instances. Since the map is a BST and is ordered on keys, it is interpreted according to following logic: suppose we have mappings (k1, n1), (k2, n2) ..... (kt, nt), t is number of nodes in the cluster, also suppose k1 < k2 < k3 < ... kt = HASH_MAX(4294967295ul) then n1 is in charge of [0, k1], n2 is in charge of (k1, k2] ...... kt is in charge of (k(t-1), kt]

• uint32_t masterID
ID of the Master

• TCPConnection* masterConnection
A shortcut for getting TCP connection to the Master which is used intensively for reporting load statistics and routing final outputs

• uint32_t numCmrNodes
Current number of C-MR-D instances

4.4.2 Public Methods

• void establishConnectionsToHosts()
Build TCP connections to all C-MR-D instances in the node address table

• uint32_t getProcessingNodeID(uint16_t opID,uint32_t hash)
Given ID of an operator and hash of a data item, look up in the operator's routing table and return ID of the C-MR-D instance the data should be routed to

• void sendRoutingTableToHosts(uint16_t opID)
Send specific operator's routing table to all C-MR-D instances except the local one

• void routeData(DataIterator* di)
Given a data item, look up in the corresponding operator's routing table for which C-MR-D instance it should be routed to according to its hash, then send it out

• void sendStatistic(uint16_t operatorID, double statistic)
Send load statistics of an operator to the Master

• void broadcastPunctuation(DataIterator* di)
Broadcast a punctuation to all C-MR-D instances including the local one

• void sendDataToMaster(DataIterator* di)
Send a data item to the Master

• TCPConnection* getConnectionToNode(uint32_t ID)
Look up in TCP connection table and return TCP connection to a C-MR-D instance with a specific ID

• TCPConnection* getConnectionToMasterNode()
Getting TCP connection to the Master

• int getNumNodes()
Get current number of C-MR-D instances

5 Conclusion and Future Work

We design and implement C-MR-D, a distributed implementation of C-MR processing model on top of C-MR-S. We try to fully utilize power of C-MR processing model while achieving scalability and high performance. In particular, we try to minimize end-to-end latency in a heterogeneous environment by making load balancing decisions based on real-time load statistics.

Currently, C-MR-D is a preliminary version. We are still exploring tradeoffs between strategies of workload partitioning and load balancing. We plan to extend the system with a dynamic membership protocol and full fault-tolerance mechanisms in the future.

References

[1] Nathan Beckman, Karthik Pattabiraman, Ugur Cetintemel, C-MR: *A Continuous MapReduce Processing Model for Low-Latency Stream Processing on Multi-Core Archcitectures*

[2] Nathan Beckman, *Thesis Proposal: An Elastic Parallel Stream Processing Model for High-Efficiency Computing on Heterogeneous Architectures*

[3] Ion Stoica et al., *Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications,* SIGCOMM'01

[4] Guiseppe DeCandia et al., *Dynamo: Amazon's Highly Available Key-Value Store,* SOSP2007